CHAPTER 1: A TUTORIAL INTRODUCTION

Let us begin with a quick introduction to C. Our aim is to show the essential elements of the language in real programs, but without getting bogged down in details, formal rules, and exceptions. At this point, we are not trying to be complete or even precise (save that the examples are meant to be correct). We want to get you as quickly as possible to the point where you can write useful programs, and to do that we have to concentrate on the basics: variables and constants, arithmetic, control flow, functions, and the rudiments of input and output. We are quite intentionally leaving out of this chapter features of C which are of vital importance for writing bigger programs. These include pointers, structures, most of C's rich set of operators, several control flow statements, and myriad details.

This approach has its drawbacks, of course. Most notable is that the complete story on any particular language feature is not found in a single place, and the tutorial, by being brief, may also mislead. We have tried to minimize this effect, but be warned.

Another drawback is that later chapters will necessarily repeat some of this chapter. We hope that the repetition will help you more than it annoys.

In any case, experienced programmers should be able to extrapolate from the material in this chapter to their own programming needs. Beginners should supplement it by writing small, similar programs of their own. Both groups can use it as a framework on which to hang the more detailed descriptions that begin in Chapter 2.

1.1 Getting Started

The only way to really learn a new programming language is by writing programs in it. And the first program is the same for all languages:

> Print upon the normal output the words
>
> hello, world

This is the basic hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy.

In C, the program to print "hello, world" is

```
main()
{
	printf("hello, world\n");
}
```

Just how to run this program depends on the system you are using. As an specific example, on Unix you must create the source program on a file whose name ends in ".c", such as *hello.c*, then compile it with the *cc* command

cc hello.c

If you haven't botched anything, such as omitting a character or misspelling something, the compilation will proceed silently, and make an executable file called *a.out*. Running that by the command

a.out

will produce

```
hello, world
```

as its output.

On other systems, the rules will be different; check with a local expert.

Exercise 1-1: Run this program on your system. Experiment with leaving out parts of the program, to see what error messages you get. □

Now for some explanations about the program itself. A C program, whatever its size, consists of one or more "functions" which specify the actual computing operations that are to be done. C functions are similar to the functions and subroutines of a Fortran program or the procedures of PL/I, Pascal, etc. `main` is such a function. Normally you are at liberty to give functions whatever names you like, but `main` is a special name — your program begins executing at the beginning of `main`. This means that every program *must* have a `main` somewhere. `main` will usually invoke other functions to perform its job, some coming from the same program, and others from libraries of previously written functions.

One method of communicating data between functions is by arguments. The parentheses following the function name surround the argument list; here `main` is a function of no arguments, indicated by `( )`. The braces `{ }` enclose the statements that make up the function. A function is invoked by naming it, followed by a parenthesized list of arguments. There is no CALL statement as there is in Fortran or PL/I. The parentheses must be present even if there are no arguments.

The line that says

```
	printf("hello, world\n");
```

is a function call, which calls a function named `printf`, with the argument `"hello, world\n"`. `printf` is a library function which prints output on the terminal (unless some other destination is specified). In this case it prints the string of characters that make up its argument.

A sequence of characters enclosed in the double quotes "..." is called a *character string*. For the moment our only use of character strings will be as arguments for printf and other functions. A string may contain any number of any characters.

The sequence \n in the string is C shorthand for the *newline character*, which when printed advances the terminal to the next line. If you leave out the \n (a worthwhile experiment), you will find that your output is not terminated properly by a line feed. The only way to get a newline character into the printf argument is with \n; if you try something like

```
printf("hello, world
");
```

the C compiler will print unfriendly diagnostics about missing quotes.

printf never supplies a newline automatically, so multiple calls may be used to build up an output line in stages. Our first program could just as well have been written

```
main( )
{
        printf("hello, ");
        printf("world");
        printf("\n");
}
```

to produce an identical output.

Notice that \n represents only a single character. There are several other "escape sequences" like \n for representing hard-to-get or invisible characters, such as \t for tab, \b for backspace, \" for the quote, and \\ for the backslash itself.

Exercise 1-2: Experiment to find out what happens when printf's argument string contains where x is some character not listed above. □

1.2 Variables and Arithmetic

The next program prints the following table of fahrenheit temperatures and their centigrade equivalents, using the formula $c=(5/9)(f-32)$.

```
  0 -17.8
 20  -6.7
 40   4.4
 60  15.6
 80  26.7
100  37.8
120  48.9
140  60.0
160  71.1
180  82.2
```

```
200  93.3
220 104.4
240 115.6
260 126.7
280 137.8
300 148.9
```

Here is the program itself.

```
/* print fahrenheit-centigrade table
	for f = 0, 20, ..., 300 */
main( )
{
	int lower, upper, step;
	float fahr, cent;

	lower = 0;	/* lower limit of temperature table */
	upper = 300;	/* upper limit */
	step = 20;	/* step size */

	fahr = lower;
	while (fahr <= upper) {
		cent = 5.0/9.0 * (fahr-32);
		printf("%4.0f %6.1f\n", fahr, cent);
		fahr = fahr + step;
	}
}
```

The first two lines

```
/* print fahrenheit-centigrade table
	for f = 0, 20, ..., 300 */
```

are a *comment,* which in this case explains briefly what the program does. Any characters between /* and */ are ignored; they may be used freely to make a program easier to understand.

In C, *all* variables must be declared before use, usually at the beginning of the function before any executable statements. If you forget a declaration, you will get a diagnostic from the compiler. A declaration consists of a *type* and a list of variables which have that type, as in

```
	int lower, upper, step;
	float fahr, cent;
```

The type int implies that the variables listed are *integers*; float stands for *floating point,* i.e., numbers which may have a fractional part. The precision of both int and float depends on the particular machine you are using; on the PDP-11, for instance, an int is a 16 bit number, that is one which lies between −32768 and +32767. A float number is a 32 bit quantity, which amounts to about seven significant digits, with exponents between −38 and +38.

C provides several other basic data types besides **int** and **float**, the most common of which are

char	character — a single byte
long	long integer
double	double precision floating point

There are also *arrays* and *structures* of these basic types, *pointers* to them, and *functions* that return them, all of which we will meet in due course.

The executable statements in the temperature conversion program begin with the assignments

```
	lower = 0;
	upper = 300;
	step = 20;
```

etc., which set the variables to their starting values. Individual statements are terminated by semicolons; no semicolon follows a brace.

To produce a table of many lines, we need a loop; this is the function of the **while** statement

```
	while (fahr <= upper) {
		...
	}
```

The condition in parentheses is tested. If it is true, the body of the loop (all of the statements enclosed in the braces { and }) is executed. Then the condition is re-tested, and if true, the body is executed again. Eventually when the test becomes false (**fahr > upper**) the loop ends, and execution continues at the statement that follows the loop. There are no further statements in this program, so it terminates.

The body of a **while** can be a single C statement, like

```
	while (...)
		printf(...);
```

or one or more statements enclosed in braces, as in the temperature converter.

The statements controlled by the **while** are indented by one tab stop so you can see at a glance what the scope of the **while** is, that is, what statements are inside the loop. The indentation emphasizes the logical structure of the program. Although C is quite permissive about statement positioning, proper indentation and use of white space are critical in making programs easy for people to read. The position of the braces is less important; we have chosen one of several popular styles. Pick a style that suits you, then use it consistently.

Most of the work gets done in the body of the loop. The centigrade temperature is computed and assigned to **cent** by the statement

```
        cent = 5.0/9.0 * (fahr-32);
```

The reason for using 5.0/9.0 instead of the simpler looking 5/9 is that in C, as in many other languages, integer division *truncates,* so any fractional part is discarded. Thus 5/9 is zero and of course so would be all the temperatures. A decimal point in a constant indicates that it is floating point, so 5.0/9.0 is 0.555..., as we want.

Why not use 32.0 instead of 32? Since fahr is a float, 32 is automatically converted to float (to 32.0) before the subtraction can be done. As a matter of style, it's wise to write floating point constants with explicit decimal points even when they have integral values; it emphasizes their floating point nature for human readers, and ensures that the compiler will see things your way too.

This example shows a bit more of how printf works. printf is actually a general-purpose format conversion function, which we'll describe completely in Chapter 7. Its first argument is a string of characters to be printed, with each % sign indicating where one of the other (second, third, ...) arguments is to be substituted, and what form it is to be printed in. In particular, %4.0f says that a floating point number is to be printed in a space at least four characters wide, with no digits after the decimal point. %6.1f describes another number to occupy six spaces, with 1 digit after the decimal point. (printf also recognizes %d for decimal integer, %o for octal, %x for hexadecimal, %c for character, %s for character string, and %% for % itself.)

Each % construction in the first argument of printf is paired with its corresponding second, third, etc., argument; they must line up properly by number and type, or you'll get meaningless answers.

By the way, printf is *not* part of the C language; there is no input or output defined in C itself. There is nothing magic about printf; it is just a useful function which is part of the standard library of routines that are normally accessible to C programs. In order to concentrate on C itself, we won't talk much about I/O until Chapter 7. In particular, we will defer formatted input until then.

Exercise 1-3: Modify the temperature conversion program to print a heading above the table. □

Exercise 1-4: Write the program which does the corresponding centigrade to fahrenheit table. □

Some Variations

As you might expect, there are plenty of different ways to write a program; let's try a variation on the temperature converter.

```
main()      /* fahrenheit-centigrade table */
{
        int fahr;

        for (fahr = 0; fahr <= 300; fahr = fahr + 20)
                printf("%4d %6.1f\n", fahr, 5.0/9.0 * (fahr-32));
}
```

This produces the same answers, but it certainly looks different. One major change is the elimination of most of the variables; only **fahr** remains, as an **int**. The lower and upper limits and the step size appear only as constants in the **for** statement, itself a new construction, and the expression that used to produce **cent** now appears as the third argument of **printf** instead of a separate assignment statement.

This last change is an instance of a quite general rule in C — in any context where it is permissible to use the value of a variable of some type, you can use an expression of that type. Since the third argument of **printf** has to be a floating point value, any floating point expression can occur there.

The **for** itself is a loop, a generalization of the **while**. If you compare it to the earlier **while**, its operation should be clear. The first part

```
fahr = 0
```

is done once, before the loop proper is entered. The condition

```
fahr <= 300
```

is evaluated, and if true, the body of the loop (here a single **printf**) is executed. Then the re-initialization step

```
fahr = fahr + 20
```

is done, and the condition re-evaluated. The loop terminates when the condition becomes false. As with the **while**, the body of the loop can be a single statement, or a group of statements enclosed in braces. The initialization and re-initialization parts can be any single expression.

The choice between **while** and **for** is arbitrary, based on what seems clearest. The **for** is usually appropriate for loops when the loop initialization and re-initialization are single statements and logically related.

Symbolic Constants

A final observation before we leave temperature conversion forever. It's bad practice to bury "magic numbers" like 300 and 20 in a program; they convey no information to someone who might have to read the program later, and they are hard to change in a systematic way. Fortunately, C provides a way to avoid such magic numbers. With the **#define** construction, you can define a *symbolic name* with a particular value at the beginning of a program, then use the name whenever the value is needed. The compiler replaces all occurrences of the name by the value.

```
#define     LOWER     0
#define     UPPER     300
#define     STEP 20

main( )     /* fahrenheit-centigrade table */
{
      int fahr;

      for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
            printf("%4d %6.1f\n", fahr, 5.0/9.0 * (fahr-32));
}
```

The replacement for the name can actually be any text at all; it is not limited to numbers. Notice that there is no semicolon at the end of a definition. Since the whole line after the defined name is substituted, there would be too many semicolons in the for.

Exercise 1-5: Verify that the construction

```
#define     LOWER     0;
```

causes an error. □

1.3 A Collection of Useful Programs

We are now going to consider a family of related programs for doing simple operations on character data. You will find that many programs are just expanded versions of the prototypes that we write here.

The standard library provides functions for reading and writing a character at a time. getchar( ) fetches the *next input character* each time it is called, and returns that character as its value. That is, after

```
c = getchar( )
```

then c contains the next character of input. The characters normally come from the terminal, but that need not concern us right now. (More on that topic in Chapter 7.)

The function putchar(c) is the complement of getchar:

```
putchar(c)
```

prints the character c on some output medium, again usually the terminal.

As with printf, there is nothing special about getchar and putchar. They are not part of the C language, but they are universally available.

File Copying

Given getchar and putchar, you can write a surprising amount of useful code without knowing anything more about I/O. The simplest example is a program which copies its input to its output a character at a time. In outline,

```
get a character
while (character is not end of file signal)
        putchar (the character just read)
        get a new character
```

Converting this into C gives

```
main( )         /* copy input to output */
{
        int c;

        c = getchar( );
        while (c != EOF) {
                putchar(c);
                c = getchar( );
        }
}
```

The relational operator != is "not equal to."

The main problem is detecting the end of the input. By convention, **getchar** returns a value which is not a valid character when it encounters the end of the input; in this way, programs can detect when they run out of input. The only complication is that there are two conventions in common use about what that end of file value really is. We have deferred the issue by using the symbolic name **EOF** for the value, whatever it might be. In practice, **EOF** will be either −1 or the null character '\0', so the program will have to be preceded by the appropriate one of

```
#define     EOF   -1
```

or

```
#define     EOF   '\0'
```

in order to work properly.

You should distinguish the value **EOF** that **getchar** returns when end of file is encountered from whatever mechanism is used by the operating system that the program runs on. For example, on Unix, end of file is implicit when a file is being read; from a terminal it can be entered by typing the character EOT ("control-D"), but on other systems this convention will probably be different.

Assignment Expression

The program for copying would actually be written more concisely by experienced C programmers. In C, any assignment statement, such as

```
c = getchar( )
```

can be used in an expression; its value is the value of the right hand side. If the assignment of a character to **c** is put inside the test part of a **while**, the file copy program can be written

```
main()      /* copy input to output */
{
	int c;

	while ((c = getchar()) != EOF)
		putchar(c);
}
```

The program gets a character, assigns it to c, and then tests if it was the end of file signal. If it was not, the body of the while is executed, printing the character. The while then repeats. When the end of the input is finally reached, the while terminates and so does main.

This version centralizes the input — there is now only one call to getchar — and shrinks the program. Nesting an assignment statement in a test is one of the places where C permits a valuable conciseness. (It's possible to get carried away and create impenetrable code, though, a tendency that we will try to curb.)

It's important to recognize that the parentheses around the assignment statement within the conditional are really necessary. The "precedence" of = is lower than that of !=, which means that the relational test != is done before the assignment =. So in the absence of parentheses, the statement

```
	c = getchar() != EOF
```

is equivalent to

```
	c = (getchar() != EOF)
```

This has the undesired effect of setting c to 0 or 1, depending on whether the character fetched was an end of file or not.

Character Counting

The next program counts characters; it is a small elaboration of the copy program.

```
main()      /* count characters in input */
{
	long nc;

	nc = 0;
	while (getchar() != EOF)
		nc++;
	printf("%ld\n", nc);
}
```

The line

```
	nc++
```

shows a new C operator. ++ means *increment* by one. You could write nc = nc + 1 but nc++ is most concise and often most efficient. There is a corresponding decrement operator --. ++ and -- can be either prefix operators (++nc) or postfix (nc++); these have different values in expressions, as we will discuss in Chapter 2, but both increment nc.

The character counting program accumulates its count in a long variable instead of an int because on a PDP-11 the maximum value of an int is 32767, and it would take relatively little input to overflow the counter if it were declared int. (In Honeywell and IBM C, long and int are synonymous and much larger.) The %ld in the printf signals a long integer.

Some early versions of C do not support long variables; in that case, use float or even double (double length float) to cope with bigger numbers. Here is the character counting program with double. We will also use a for statement instead of a while, to illustrate an alternative construction.

```
main()      /* count characters in input */
{
        double nc;

        for (nc = 0; getchar() != EOF; nc = nc + 1)
                ;
        printf("%f\n", nc);
}
```

Some versions of C do not permit ++ and -- to be applied to float or double. In that case, you must write out the increment as we did here.

printf uses %f for both float and double. You could also use %.0f to suppress printing the non-existent fraction part.

The body of the for loop here is *empty,* because all of the work is done in the test and re-initialization parts. But the grammatical rules of C require that a for statement have a body. The isolated semicolon, technically a *null statement,* is there to satisfy that requirement. We put it on a separate line to make it more visible.

Before we leave the character counting program, observe that if the input contains no characters, the while or for test fails on the very first call to getchar, and so the program produces zero, the right answer. This is an important observation. One of the nice things about while and for is that they test at the *top* of the loop, before proceeding with the body. If there is nothing to do, nothing is done, even if that means never going through the loop body. Programs should act intelligently when handed input like "no characters". The while and for help ensure that they do reasonable things with extreme cases.

Line Counting

The next program counts *lines* in its input. Input lines are assumed to be terminated by the newline character \n that has been religiously appended to every line written out. The program is again simple and familiar.

```
main()        /* count lines in input */
{
      int c, nl;

      nl = 0;
      while ((c = getchar()) != EOF)
            if (c == '\n')
                  nl++;
      printf("%d\n", nl);
}
```

The body of the while now consists of an if, which in turn controls the increment nl++. if tests the parenthesized condition, and if it is true, does the statement (or group of statements in braces) that follows. We have again indented to show what is controlled by what.

The double equals sign == is the C notation for "is equal to" (like Fortran's .EQ.); it is pronounced "equals". A separate symbol is used to distinguish the equality test from the single = used for assignment. Since assignment is about twice as frequent as equality testing in typical C programs, it's appropriate that the operator be half as long.

Exercise 1-6: Write a program to count blanks, tabs, and newlines. □

Word Counting

The fourth in our series of useful programs counts lines, words, and characters, with the loose definition that a word is any sequence of characters that does not contain a blank, tab or newline.

```
#define    YES  1
#define    NO   0

main()     /* count words, lines, chars in input */
{
      int c, nc, nl, nw, inword;

      inword = NO;
      nc = nl = nw = 0;
      while ((c = getchar()) != EOF) {
            nc++;
            if (c == '\n')
                  nl++;
            if (c == ' ' || c == '\n' || c == '\t')
                  inword = NO;
            else if (inword == NO) {
                  inword = YES;
                  nw++;
            }
      }
      printf("%d %d %d\n", nl, nw, nc);
}
```

Logically, this is a more complicated program. Every time the program makes the transition from not being in a word to being in a word, it counts one more word. The variable inword records which state the program is in; initially it is "not in a word", which is assigned the value NO. We prefer the symbolic constants YES and NO to the literal values 1 and 0 because they make the program more readable. Of course in a program as tiny as this, it makes little difference, but in larger programs, the increase in clarity is well worth the modest extra effort to write it this way originally. You'll also find that it's easier to make extensive changes in programs where numbers appear only as symbolic constants.

The line

```
nc = nl = nw = 0;
```

sets all three variables to zero. This is not a special case, but a consequence of the fact that the assignment statement has a value. It's really as if we had written

```
nc = (nl = (nw = 0));
```

The operator | means OR, so the line

```
if (c == ' ' || c == '\n' || c == '\t')
```

says "if c is a blank *or* c is a newline *or* c is a tab ...". (The character \t is a tab; it is best written this way so it is visible on a listing.) There is also && for AND. Expressions connected by && and | are evaluated left to right, and it is

||

guaranteed that evaluation will stop as soon as the answer is known. Thus if c contains a blank, there is no need to test whether it contains a newline or tab, so these tests are *not* made.

The example also shows the C **else** statement, which specifies an alternative action to be done if the condition part of an **if** statement is false. One and only one of the two statements associated with an **if-else** is done. Either statement can in fact be quite complicated. In the word count program, the one after the **else** is another **if**.

Exercise 1-7: Write a program which prints only the words in its input, one per line. All non-alphabetic characters should be discarded. As a matter of design, what should happen to digits? What about contractions like don't? □

1.4 Arrays

Let us write a program to count the number of occurrences of each digit, of white space (blank, tab, newline), and all other characters. This is artificial, of course, but it permits us to illustrate several aspects of C in one program.

There are twelve categories of input, which is too many to use separate variables for each, so we must at least use an array to hold the number of occurrences of digits. Here is one version of the program:

```
main()      /* count digits, white space, others */
{
      int c, i, nwhite, nother, ndigit[10];

      nwhite = nother = 0;
      for (i = 0; i < 10; i++)
            ndigit[i] = 0;

      while ((c = getchar()) != EOF)
            if (c >= '0' && c <= '9')
                  ndigit[c-'0']++;
            else if (c == ' ' || c == '\n' || c == '\t')
                  nwhite++;
            else
                  nother++;

      printf("digits = ");
      for (i = 0; i < 10; i++)
            printf("%d ", ndigit[i]);
      printf("\nwhite space = %d, other = %d\n", nwhite, nother);
}
```

The declaration

```
      int ndigit[10]
```

declares ndigit to be an array of 10 integers. Array subscripts start at zero in C (rather than 1 as in Fortran or PL/I), so the elements are **ndigit[0]**, **ndigit[1]**, ... **ndigit[9]**. This is reflected in the **for** loops which initialize and print the array.

A subscript can be any integer expression; this includes as special cases integer variables like **i**, and integer constants.

This particular program relies heavily on the properties of the character representation of the digits. For example, the test

```
c >= '0' && c <= '9'
```

for a digit and the character-to-integer conversion

```
c - '0'
```

work only if the digits are ordered and if there is nothing but digits between '0' and '9'. Fortunately, this is universally true.

char variables and constants are essentially identical to **int**'s in arithmetic contexts. Thus **c−'0'** is an integer expression with a value between 0 and 9 corresponding to the character '0' to '9' stored in **c**. By definition, arithmetic involving **char**'s and/or **int**'s converts everything to **int** before proceeding, so things work out quite naturally and conveniently. In fact, single **char**'s are often just declared **int**'s.

The pattern

```
if (condition)
        statement
else if (condition)
        statement
else
        statement
```

occurs frequently in programs as a way to express a multi-way decision; here the decision is digit, white space, or other. The code is simply read from the top until some *condition* is satisfied; at that point the corresponding *statement* part is executed, and the entire construction exited. (Of course *statement* can be several statements enclosed in braces.) If none of the conditions is satisfied, the *statement* after the final **else** is done. If that is omitted (as in the word count program), no action takes place. There can be an arbitrary number of

```
else if (condition)
        statement
```

groups between the initial **if** and the final **else**. As a matter of style, it is advisable to format this construction as we have shown, so that long decisions do not march off the right side of the page.

The **switch** statement, to be discussed in Chapter 3, provides another way to write a multi-way branch that is particularly suitable when the condition being tested is simply whether some integer or character expression matches one of a set of constants. We will present a **switch** version of this program in Chapter 3.

Exercise 1-8: Write a program to remove any trailing blanks or tabs from each line of input. □

1.5 Functions

A function is just a way to encapsulate some part of a computation in a black box, which can then be used without worrying about its innards. So far we have used only functions that have been provided for us: printf, getchar and putchar. Now it's time to write a few of our own.

To illustrate the points, let's write a program to print the longest line in the input. The basic outline is simple enough:

```
while (there's a new line)
      if (it's longer than the previous longest)
            save it and its length
print longest line
```

This makes it clear that the program divides naturally into pieces. One piece gets a new line, another tests it, another saves it, and the rest control the process.

Since things divide so nicely, it would be well to write them that way too; that is the purpose of functions. Accordingly, let us first write a separate function to fetch the next *line* of input; this is a generalization of getchar. Since we want to make the function useful in other contexts, we'll try to make it as flexible as possible. Let us say that the newline at the end of the line will be trimmed off, so that the resulting line can be used as a character string. At the minimum, getline has to return a signal about possible end of file; a more generally useful design would be to return the length of the line, or −1 if end of file is encountered.

When we find a longer line than the previous longest, it must be saved somewhere. This suggests a second function copy, to copy the new line to a safe place.

Finally, we need a main program to control getline and copy. Here is the whole program at once, so you can see what it looks like.

```
#define      MAXLINE    1000 /* maximum input line size */

main()       /* find longest line */
{
        int n, max;
        char line[MAXLINE], save[MAXLINE];

        max = -1;  /* longest length seen so far */
        while ((n = getline(line, MAXLINE)) >= 0)
                if (n > max) {
                        max = n;
                        copy(line, save);
                }
        if (max > -1)        /* there was a line */
                printf("%s\n", save);
}

getline(s, lim)         /* read line into s; return length */
char s[];
int lim;        /* maximum line length, including \0 */
{
        int i, c;

        for (i = 0; i < lim-1 && (c = getchar()) != '\n' && c != EOF; i++)
                s[i] = c;
        s[i] = '\0';
        if (c == EOF)
                return(-1);
        else
                return(i);
}

copy(s1, s2) /* copy s1 to s2; assumes s2 big enough */
char s1[], s2[];
{
        int i;

        for (i = 0; (s2[i] = s1[i]) != '\0'; i++)
                ;
}
```

Each function has the same form:

```
name(argument list, if any)
argument declarations, if any
{
        declarations
        statements
}
```

The functions can appear in any convenient order, in one source file or in several. (Whatever the allocation to source files, the definition of EOF must be accessible to getline.) Of course if the input appears in several files, you will have to do more work to compile and load it than if it all appears on one file, but that is an operating system matter more than a property of the C language. For the moment, we will assume that the three functions are all in one file.

main and getline communicate through both a pair of arguments and a returned value. In getline, the arguments have to be declared appropriately; this is done by the lines

```
char s[ ];
int lim;
```

which specify that the first argument is an array of indeterminate length, and the second is an integer. The declaration of arguments goes between the function argument list and the opening left brace. The names used by getline for the arguments are purely local to getline, and not accessible to anyone else.

getline uses a return statement to send a value back to the caller, just as in PL/I. Any expression may occur in the parentheses. A return with no expression causes control, but no value, to be returned to the caller. The same is true of "falling off the end" of a function, as in copy.

getline puts the character \0 (the *null character,* whose value is zero) at the end of the array it is creating, to mark the end of a string of characters. This convention is followed throughout C. For example, when a string constant like

```
"hello, world\n"
```

is written in a C program, the compiler terminates the array of characters representing that string with a \0 so that functions such as printf can detect the end.

If you examine copy closely, you will discover that it relies on the fact that its input argument s1 is terminated by \0, and it copies this character onto the output argument s2.

Exercise 1-9: Write a program to "fold" long input lines at the first blank or tab before the *n*th column of input. Make sure that n is a parameter in your program. Do something intelligent if there are no blanks or tabs before the specified column. □

1.6 Arguments — Call by Value

One aspect of function usage can trap programmers used to other languages, particularly Fortran and PL/I. In C, all function arguments are passed "by value." This means that the called function is given the values of its arguments in temporary variables (actually on a stack) rather than their addresses. This leads to quite different properties than are seen with "call by reference" languages like Fortran and PL/I, where the called routine is handed the address of the argument variable, not its value.

The main distinction is that in C the called function *cannot* alter the original arguments in the calling function; it can only alter its private, temporary copy.

Call by value is generally an asset, however, not a liability. Arguments can be treated as conveniently initialized variables in the called routine. For example, consider the function log2(n), which returns the base 2 logarithm of n (that is, the number of bits needed to hold n), where n must be a positive int.

```
log2(n)       /* base 2 log of n */
int n;
{
        int log;

        log = 1;
        while ((n = n / 2) > 0)
                log++;
        return(log);
}
```

The argument n is used as a temporary variable, which is repeatedly divided by two until it becomes zero; there is no need for another variable in log2 to hold the same value. And whatever is done to n inside log2 has no effect on the argument that log2 was originally called with.

With call by value, array arguments are processed without special effort, since the name of an array is in effect the address of the first element, and once you know the address of something, it's easy to work your will on it. We have already done this with functions like getline and copy that store into character arrays.

It is possible to arrange for a function to modify a variable in a calling routine when necessary. The caller must provide the *address* of the variable to be set; and the callee must declare it to be a pointer and reference it indirectly. We will cover this in detail in Chapter 5.

1.7 Scope

The variables in main (line, save, etc.) are private or *local* to main; because they are declared within main, no other function can have direct access to them. The same is true of the variables in the other functions; for example, the variable i in getline is unrelated to the i in copy. Each local variable in a routine comes into existence only when the function is called, and *disappears* when the function is exited. Accordingly, local variables have no memory from one call to the next and must be explicitly initialized upon each entry. It is for this reason that local variables are also known as *automatic* variables, following terminology in other languages. (Chapter 4 discusses the static storage class, in which local variables do retain their values between function invocations.)

There is a need to share data between routines, however, so C provides two ways to achieve it. The first is function arguments and return values, which we have used in all our examples so far. The second is a way to define variables which are *external* to all functions, that is, global variables which can be accessed by name by any function that cares to. (This mechanism is rather like Fortran COMMON or PL/I EXTERNAL.) External variables remain in permanent existence, rather than appearing and disappearing as functions are called and exited.

To make a variable external, we have to *define* it outside of any function, and make an extern declaration in each function that wants to use it. To make the discussion concrete, let us rewrite the longest-line program with line, save and max as external variables. This requires changing the calls, declarations, and bodies of getline and copy.

```
#define    MAXLINE    1000 /* maximum input line size */

char  line[MAXLINE];      /* input line */
char  save[MAXLINE];      /* longest line saved here */
int   max;  /* length of longest line seen so far */

main()        /* find longest line */
{
        int n;

        max = -1;
        while ((n = getline()) >= 0)
                if (n > max) {
                        max = n;
                        copy();
                }
        if (max >= 0)
                printf("%s\n", save);
}

getline()     /* specialized version */
{
        int i, c;
        extern char line[];

        for (i = 0; i<MAXLINE-1 && (c = getchar()) != '\n' && c != EOF; i++)
                line[i] = c;
        line[i] = '\0';
        if (c == EOF)
                return(-1);
        else
                return(i);
}

copy()        /* specialized version */
{
        int i;
        extern char line[], save[];

        for (i = 0; (save[i] = line[i]) != '\0'; i++)
                ;
}
```

Roughly speaking, any function that wishes to access an external variable must contain an **extern** declaration for it, although this can sometimes be done by context instead of explicitly. The declaration is the same as others, except for the added keyword **extern**. Furthermore, there must appear a *definition* of these variables which is external to all functions, as in the first lines of the example above.

In certain circumstances, the **extern** declaration can be omitted: if the external definition for a variable occurs *before* its use in some function, then there is no need for an **extern** declaration. The declarations in **getline** and **copy** are redundant.

There is a tendency to make everything in sight an **extern** variable because it appears to simplify communications — argument lists are short and variables are always there when you want them. But external variables are always there, even when you don't want them. This style of coding is fraught with peril, though, since it leads to programs whose data connections are not at all obvious — variables can be changed in unexpected and even inadvertent ways. The second version of the longest line program is inferior to the first, partly for these reasons, and partly because we have destroyed the generality of two quite useful functions by wiring into them the names of the variables they will manipulate.

1.8 Summary

At this point we have covered what might be called the conventional core of C. Variables and constants are the basic objects that a program processes. Each variable must be declared to be of one of the types that C supports. A variable may be an array, with subscripts running from zero to one less than the size specified in the declaration. Arithmetic operators include the usual +, −, *, and /, and the increment and decrement operators ++ and −−. Control flow operations include **if**, perhaps with an **else** part; loops with **while** or **for**; and statements grouped with braces. Functions communicate with arguments, returned values, and external variables.

With this handful of building blocks, it's possible to write useful programs of considerable size, and it would probably be a good idea if you paused long enough to do so. The exercises that follow are intended to give you some suggestions for programs of about the level of complexity that we have written here. All can be handled with small programs, no more than say 20 lines.

Exercise 1-10: Write a program to replace all strings of blanks and tabs by a single blank. □

Exercise 1-11: Write a program to replace each tab by the sequence > and each backspace by <. □ not clear what it is

Exercise 1-12: Write a program to remove all comments from a C program. Don't forget to handle quoted strings properly. □